---

title: Confidential Store description: Understanding the key primitives required to build great SUAPPs keywords: - contract - deploy - suave - solidity - confidential - private - store

---

Using the Confidential Store

If you've followed along from the previous tutorial, you'll have deployed a simple contract with onchain and offchain functions that use the SUAVE-STD library to emit logs onchain that come from offchain compute results.

Now it's time to look at another key primitive often required to build powerful Suapps: the confidential data store.

In particular, we'll consider how to store a private key confidentially, and then use that key to sign transactions intended for other chains. This pattern is useful when you want the results of your offchain computation to cause a transaction on another chain, and it finds application in everything from Uniswap v4 hooks to NFTs for concert tickets.

Import SUAVE-STD

We're going to want to use more of the functionality offered by SUAVE-STD in order to both store our private key, and then retrieve it and use it to sign a transaction intended for another domain.

Create a new file called ConfidentialStore.sol and begin by importing the supporting contracts and libraries we need:

```solidity // SPDX-License-Identifier: Unlicensed pragma solidity ^0.8.8;

import "suave-std/Suapp.sol"; import "suave-std/Context.sol"; import "suave-std/Transactions.sol"; import "suave-std/suavelib/Suave.sol"; ```

If you look through each of these imports, you'll see that:

1. Suapp.sol gives us the ability to easily emit logs from offchain computations onchain.
2. Context.sol allows any function to determine whether there are confidential inputs being passed along with the function call. We will use this to pass our private key to our Suapp without revealing what it is.
3. Transactions.sol helps decode/encode transactions from/to other domains.
4. Suave.sol contains all the precompiles which make up the MEVM that runs in each Kettle, along with the addresses they're deployed to.

## Store Keys Confidentially

Next, add the logic you'll need to store a private key confidentially with this Suapp:

```solidity contract ConfidentialStore is Suapp { Suave.DataId signingKeyRecord; string public PRIVATE_KEY = "KEY";

function updateKeyOnchain(Suave.DataId _signingKeyRecord) public {
    signingKeyRecord = _signingKeyRecord;
}

function registerPrivateKeyOffchain() public returns (bytes memory) {
    bytes memory keyData = Context.confidentialInputs();

    address[] memory peekers = new address[](1);
    peekers[0] = address(this);

    Suave.DataRecord memory record = Suave.newDataRecord(0, peekers, peekers, "private_key");
    Suave.confidentialStore(record.id, PRIVATE_KEY, keyData);

    return abi.encodeWithSelector(this.updateKeyOnchain.selector, record.id);
}

} ```

The confidential store is a key value store, and the convention is to store "data records" as the values that are keyed by "data IDs". The SUAVE library helps abstract this so you can just call Suave.DataId etc.

A newDataRecord expects four values:

1. The "decryption condition" - this is an artifact which will be removed in later versions of suave-geth.
    1. Set to 0 for now.
2. The "allowed peekers" - this determines who can "get" data associated with the DataId.
3. The "allowed stores" - this determines who can "set" data associated with the DataId.
    1. In this example we set the allowedPeekers == allowedStores == an array of 1 address, which is set to address(this). That is, only this contract can get the private key we're storing, or set it to something else.
4. The "data type" - a string which specifies the type of data being stored.
    1. In this case, it is set to "private_key".

We are also following the same pattern as previous tutorials, with an offchain function that does the heavy lifting (creating the data record for the private key we want to store), which then returns a callback to an onchain function that (rather than emitting an event)

updates the DataId by which the private key may be fetched by this specific contract.

Anyone can call this function (you may want to change that) and, if they pass in a private key in the confidentialInputs field of their Confidential Compute Request (CCR), then the Kettle which processes that CCR will set the private key in its store according to the logic above, all without revealing what that key is.

So, let's compile the contract, deploy it, and send the CCR that will store a private key!

bash forge build bash suave-geth spell deploy ConfidentialStore.sol:ConfidentialStore

If you built suave-geth from source, you may need to specify the whole path:

bash ./<path_to_suave-geth>/build/bin/suave-geth spell deploy ConfidentialStore.sol:ConfidentialStore

Now we can send the private key as a confidential input when we call registerPrivateKeyOffchain:

bash suave-geth spell conf-request --confidential-input b71c71a67e1177ad4e901695e1b4b9ee17ae16c6668d313eac2f96dbcda3f291 <your_new_contract_address> 'registerPrivateKeyOffchain()'

We're passing in private key with the --confidential-input, which - in this case - is a hex-encoded string, without the 0x. The logs printed to your console should pick up that you are passing in a confidential input:

bash INFO [04-04|11:35:27.419] Running with local devchain settings INFO [04-04|11:35:27.419] Confidential input provided input="[98 55 49 99 55 49 97 54 55 101 49 49 55 55 97 100 52 101 57 48 49 54 57 53 101 49 98 52 98 57 101 101 49 55 97 101 49 54 99 54 54 54 56 100 51 49 51 101 97 99 50 102 57 54 100 98 99 100 97 51 102 50 57 49]" INFO [04-04|11:35:27.419] Contract at address 0xd594760B2A36467ec7F0267382564772D7b0b73c INFO [04-04|11:35:27.419] Sending offchain confidential compute request kettle=0xB5fEAfbDD752ad52Afb7e1bD2E40432A485bBB7F INFO [04-04|11:35:27.425] Hash of the result onchain transaction hash=0xa1094c169d420fb0115df3be6115711b3ed54b2c2fa1fcaa28f22cdb8a69c8ce INFO [04-04|11:35:27.425] Waiting for the transaction to be mined... INFO [04-04|11:35:27.529] Transaction mined status=1 blockNum=2

There's nothing else to see in the logs just yet, as we're not emitting events from the updateKeyRecordOnchain function (thouh you can modify that yourself if you like).

## Sign a Tx with your Private Key

Let's now add the other important piece we need for this contract, which is using the private key we just stored to sign a transaction on another domain (in this case, our local ChannelSplitterNode, though you can modify this to be Eth L1 if you like):

```solidity event TxnSignature(bytes32 r, bytes32 s);

function onchain() public emitOffchainLogs {}

function offchain() public returns (bytes memory) { bytes memory signingKey = Suave.confidentialRetrieve(signingKeyRecord, PRIVATE_KEY);

Transactions.EIP155Request memory txnWithToAddress = Transactions.EIP155Request({
    to: address(0x0000000000000000000000000000000DeaDBeef),
    gas: 1000000,
    gasPrice: 500,
    value: 1,
    nonce: 1,
    data: bytes(""),
    chainId: 1337
});

Transactions.EIP155 memory txn = Transactions.signTxn(txnWithToAddress, string(signingKey));
emit TxnSignature(txn.r, txn.s);

return abi.encodeWithSelector(this.onchain.selector);

} ```
```

By now, this onchain-offchain pattern should be becoming more familiar. Offchain, we do the heavy lifting of constructing the transaction object, which we then emit in the logs of an event onchain by returning a callback to the onchain function.

The theory here is that any searcher or other service could listen to logs from TxnSignature events in this contract on SUAVE and submit them as part of their bundles to block builders for Ethereum L1. However, you can also use the Gateway pattern discussed in the next tutorial to call your preferred RPC provider yourself, such that you need not rely on these events being detected.

Recompile and redeploy your contract, and call the offchain function to see this all in action:

bash forge build bash suave-geth spell deploy ConfidentialStore.sol:ConfidentialStore

bash suave-geth spell conf-request <your_new_contract_address> 'offchain()'

You should see something like this printed to your terminal:

bash INFO [04-02|14:36:17.902] Running with local devchain settings INFO [04-02|14:36:17.903] No confidential input provided, using empty string INFO [04-02|14:36:17.903] Contract at address 0x9a151AA453329f3cdf04D8e4e81585A423f7fC25 INFO [04-02|14:36:17.903] Sending offchain confidential compute request kettle=0xB5fEAfbDD752ad52Afb7e1bD2E40432A485bBB7F INFO [04-02|14:36:17.909] Hash of the result onchain transaction hash=0xaf99b4460039486e93e64da870b890a3cf19dab728c16d89f109e058a8323f9d INFO [04-02|14:36:17.909] Waiting for the transaction to be mined... INFO [04-02|14:36:18.013] Transaction mined status=1 blockNum=10 INFO [04-02|14:36:18.117] Logs emitted in the onchain transaction numLogs=1 INFO [04-02|14:36:18.117] Log emitted name=TxnSignature(bytes32,bytes32) r="[238 188 250 192 222 246 219 86 73 208 174 107 82 237 59 139 161 245 198 196 40 88 141 241 37 70 17 19 186 140 103 73]" s="[93 94 26 175 160 201 100 180 60 37 27 106 82 93 73 87 41 104 242 206 188 88 104 197 139 204 146 129 185 160 117 5]"

The log at the bottom which returns both r and s values proves that the key you stored in the Confidential Data Store was both correctly stored, and correctly used to sign the transaction.

Congratulations! You've just begun to master the confidential store. The applications that can be built from this kind of extensive. We're excited to see what you build...